Chapter 5 Container Views

1. Introduction

The `body` property of `View` protocol can return only a single object conforming to the `View` protocol. However, for any real iOS app it is necessary to more than one object on the screen. SwiftUI provides us with container views to combine multiple views into a single object that can be returned by the `body` property. We have already come across these container views, such as `VStack`, `HStack`, `List`, and `Navigation` view.

In this chapter, we will concentrate on `NavigationView`, `TabView`, `Group`, and `ScrollView`.

Code for this chapter is available on GitHub:
https://github.com/mhuq1138/swiftui-chapter-5-container-views

1. Simple Navigation View

In the previous chapter, we have considered `List` view that can used to create a table with a single column that can be populated using static or dynamic data. However, `List` view alone cannot provide us with all the functions associated with UITableView. It becomes necessary to introduce navigation. SwiftUI provides us with the `NavigationView` for this purpose. It is a view for presenting a stack of views representing a visible path in a navigation.

SwiftUI's `NavigationView` plays more or less the same role as UINavigationController in UIKit. It provides us with a navigation bar at the top of the screen, where we can add a title and navigation bar items. A primary role of `NavigationView` is to handle navigation between views.

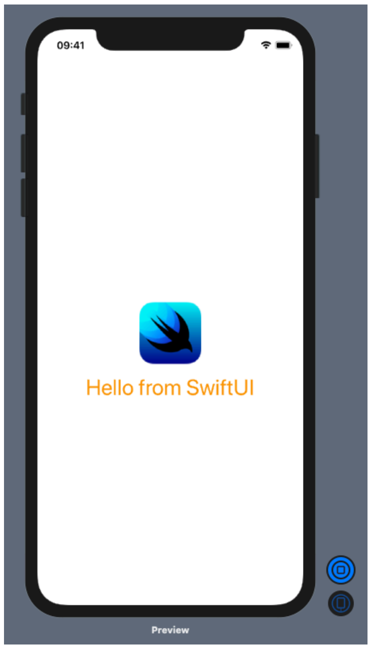
In this section, we use a simple example to demonstrate the functions of `NavigationView`. With that in mind, we create a project "Simple NavigationView Demo"

SwiftUI provides us with a single initializer to create instance of `NavigationView`:

```
init(@ViewBuilder content: () -> Content)
```

The single parameter `content` applies `ViewBuilder` to create the content of the `NavigationView` from the objects provided. In our project, we are going to place and an image in the `NavigationView`:

```
struct ContentView: View {
    var body: some View {
        NavigationView{
            VStack{
                Image("swiftui-96x96")
                Text("Hello from SwiftUI")
                   .font(.largeTitle)
                   .foregroundColor(.orange)
            }
        }
    }
}
```



In the live preview, we see that the content has been pushed down from the center of the screen. This is due to presence of the navigation bar. In iOS 13, the navigation bar is transparent when there is navigation bar title and when there is a bar title using the `displayMode` is `.large`.

Our next task is to add a navigation bar title using a view modifier. Here we would consider two of them:

```
func navigationBarTitle<S>(_ title: S) -> some View where S : StringProtocol
```

```
func navigationBarTitle(_ title: Text, displayMode: NavigationBarItem.TitleDisplayMode)
                                                                          -> some View
```

The displayMode in the second initializer is an enum with three cases:

`case automatic`

Inherits the display mode from the previous navigation item.

`case inline`

Display the title within the standard bounds of the navigation bar.

`case large`

Display a large title within an expanded navigation bar.

Let us first apply the first view modifier:

```
struct ContentView: View {
    var body: some View {
        NavigationView{
            VStack{
                Image("swiftui-96x96")
                Text("Hello from SwiftUI")
                   .font(.largeTitle)
                   .foregroundColor(.orange)
            }.navigationBarTitle("Welcome")
        }
    }
}
```

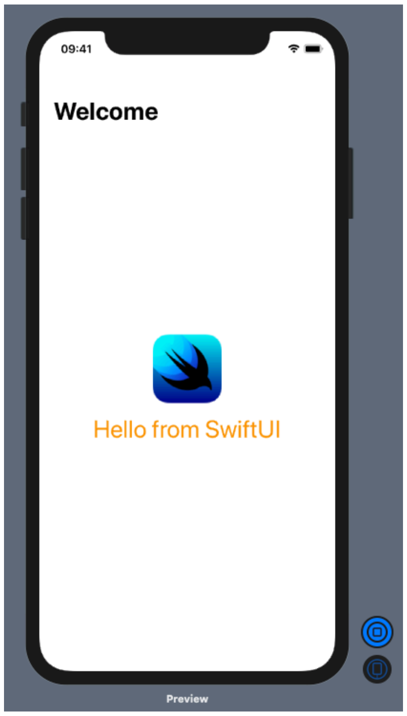
Note that the `navigationBarTitle` modifier is not applied to the `NavigationView`, it is applied to the content. But it has no effect on the content. This is because the `navigationBarTitle` modifier is one of the view modifiers that don't modify the view it is applied to. What it is actually doing is to tell SwiftUI to look around and find the current `NavigationView` in which it is residing. When it does, it will change the title to what we want.

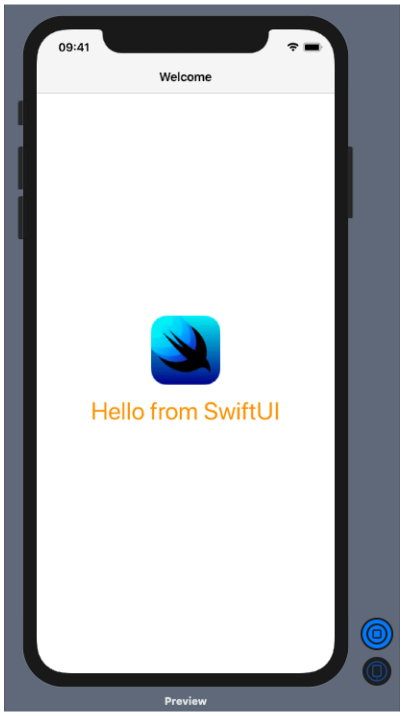
Before showing this in preview, let us choose the second view modifier:

```
struct ContentView: View {
    var body: some View {
        NavigationView{
            VStack{
                Image("swiftui-96x96")
                Text("Hello from SwiftUI")
                   .font(.largeTitle)
                   .foregroundColor(.orange)
            }.navigationBarTitle("Welcome", displayMode: .inline)
        }
    }
}
```

We have chosen the `.inline` mode, choosing one of the other two will yield the same outcome as the first view modifier.

Let us now compare the two in preview live:





We get the left screen shot when we use the first view modifier or the second view modifier with `displayMode` `.automatic` or `.larger`. The right screen shot is for the second view modifier with the `.inline` `displayMode`.

Next we want to introduce navigation to another view. We start by adding the following `View` to our project:

```
struct DetailView: View {
    var body: some View {
        Text("Hello from DetailView")
            .font(.largeTitle)
            .foregroundColor(.white)
            .padding()
            .background(Color.red)
    }
}
```

We perform navigation between views by using the `NavigationLink` button with the declaration:

```
struct NavigationLink<Label, Destination> where Label : View, Destination : View
```

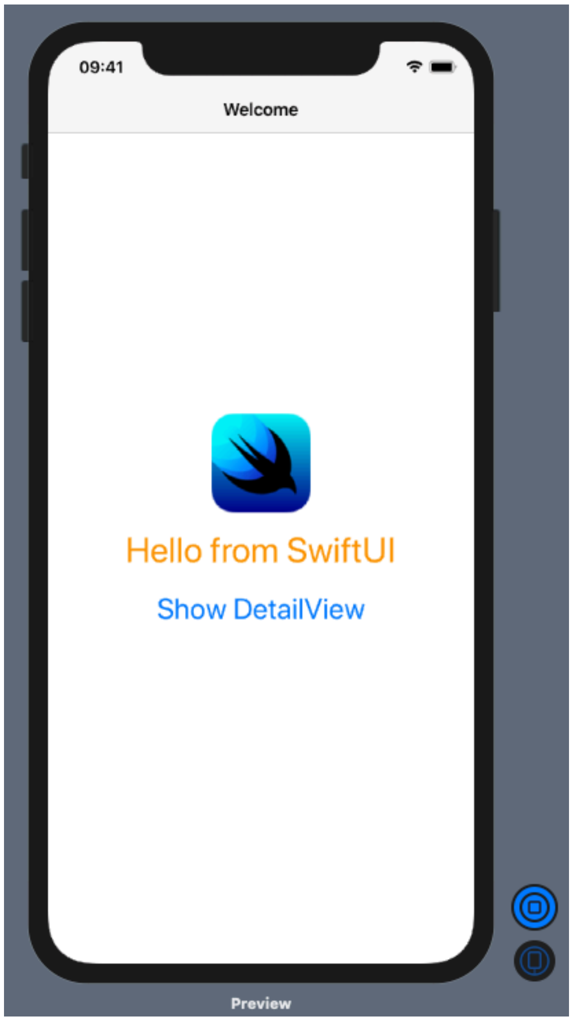
There are several initializers for the `NavigationLink` button. We will start with the following initializer:

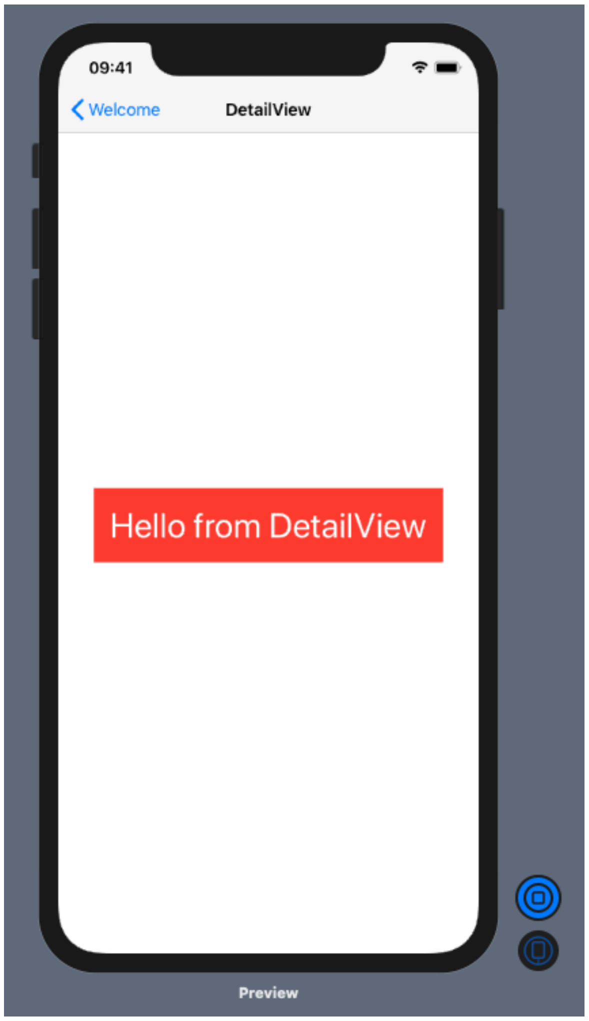
```
init(destination: Destination, @ViewBuilder label: () -> Label)
```

We now modify the code in `ContentView` to add navigation:

```
struct ContentView: View {
    var body: some View {
        NavigationView{
            VStack{
                Image("swiftui-96x96")
                Text("Hello from SwiftUI")
                   .font(.largeTitle)
                   .foregroundColor(.orange)

                NavigationLink(destination: DetailView()
                    .navigationBarTitle("DetailView", displayMode: .automatic)){
                    Text("Show DetailView")
                        .font(.title)
                        .foregroundColor(.blue)
                }.padding(20)
            }.navigationBarTitle("Welcome", displayMode: .inline)
        }
    }
}
```

In live preview (or in a simulator), when we tap on the navigation link, the `DetailView` is presented





See how the back button has appeared magically when the DetailView is displayed. When we tap the back button, we go back to the Welcome screen. However, if we tap on the navigation link again, it does not work

Warning!

This is a serious bug in iOS 13. Until Apple decides to remove the bug, we need to find a way around.

Since we not in production, we will ignore the bug. For us, it is only a minor inconvenience. We test again, we have to relaunch the preview or the simulator.

NavigationView can walk with a stack of views. In our project, we can navigate from `DetailView` to a third view and walk back from the third view to the `DetailView` and then to the welcome . So, we add the following view to our project:

```swift
struct DetailView2: View {
    var body: some View {
        Text("Hello from DetailView2")
        .font(.largeTitle)
        .foregroundColor(.white)
        .padding()
        .background(Color.orange)
      }
}
```

For navigation from the DetailView, we wish to use the following initializer:

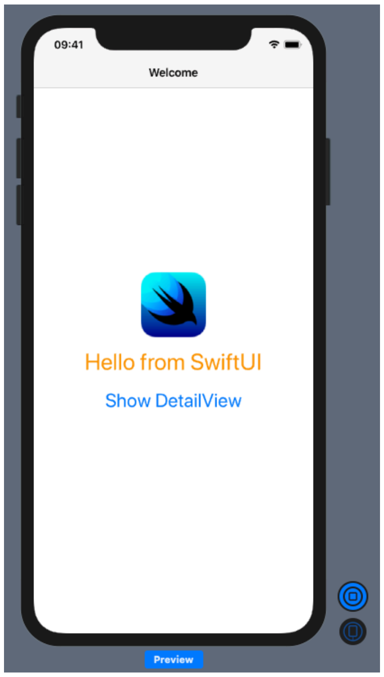
```swift
init(destination: Destination, isActive: Binding<Bool>, @ViewBuilder label: () -> Label)
```

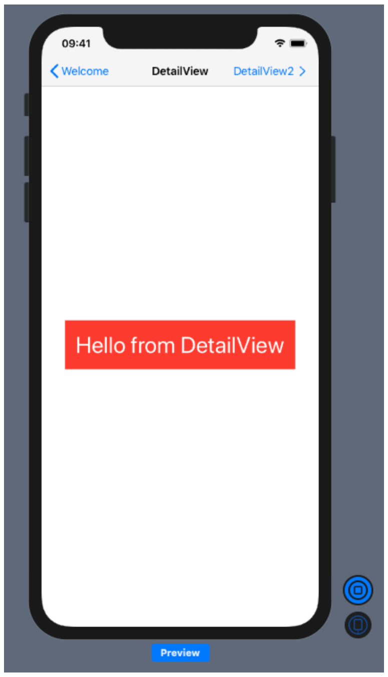
Note that this would be trigger when the binding parameter `isActive` is set `true`. Then we go ahead modify the `DetailView` code:

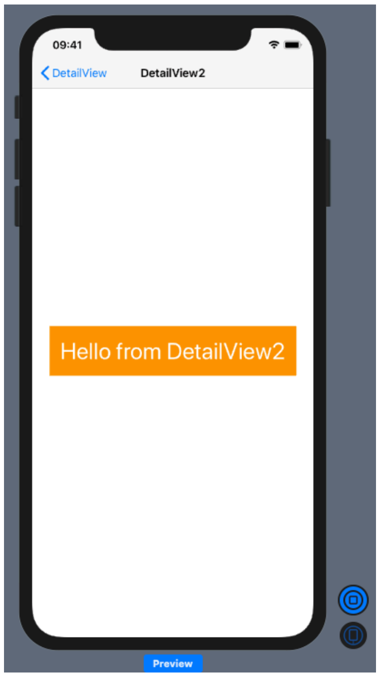
```swift
struct DetailView: View {
    @State var push = false
    var body: some View {
        VStack{
            Text("Hello from DetailView")
               .font(.largeTitle)
               .foregroundColor(.white)
               .padding()
               .background(Color.red)
            NavigationLink(destination: DetailView2().navigationBarTitle("DetailView2",
                                                    displayMode: .automatic), isActive: $push){
                   EmptyView()
            }
        }.navigationBarItems(trailing: Button(action: {self.push = true}){
            HStack{
                Text("DetailView2")
                Image(systemName: "chevron.right")
            }
        })
    }
}
```

The `NavigationLink` has `EmpyView()` as its `Label`. So, it has no visual appearance. It is triggered by the button we have places as `NavigationBarItem`, which sets the state variable `push` `true` triggering the link.

Then live preview:







When we tap `Show DetailView`, the `DetailView` appears (middle screen shot). Tapping the bar button DetailView2> in the navigation bar to go to `DetailView2` (right screen shot). We can go back to the welcome screen by successive tap on the back button. In this case, there is no bug. We repeat the above steps starting with `Show DetailView`. The bug appears when we go back one step and try to go to the next view down the line.

2. List with NavigationView

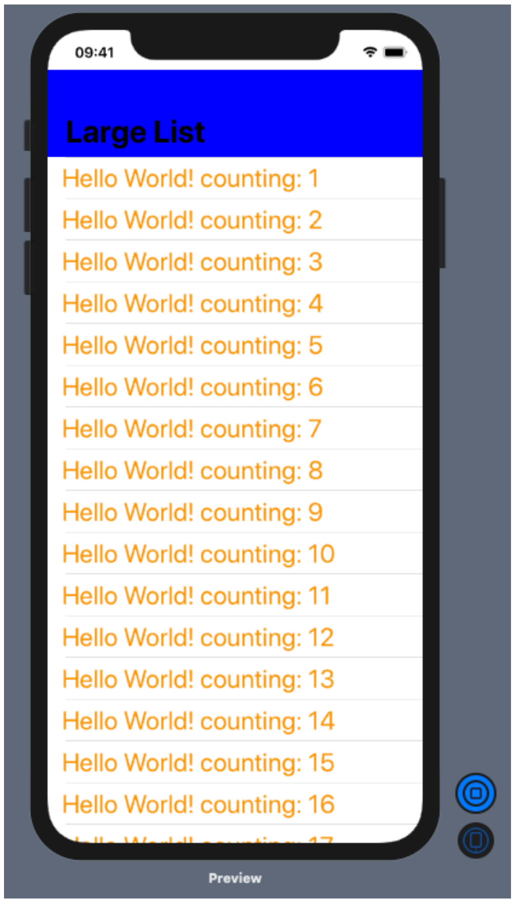
We create a project "List with NavigationView Demo" with the code:

```
struct ContentView: View {
    var body: some View {
        NavigationView{
            List(1...50, id: \.self){i in
                Text("Hello World! counting: \(i)")
                    .font(.title)
                    .foregroundColor(.orange)
            }.navigationBarTitle("Large List ")
        }
    }
}
```

Note that we can change the default navigation bar background by adding the following code to `SceneDelegate.swift` file:

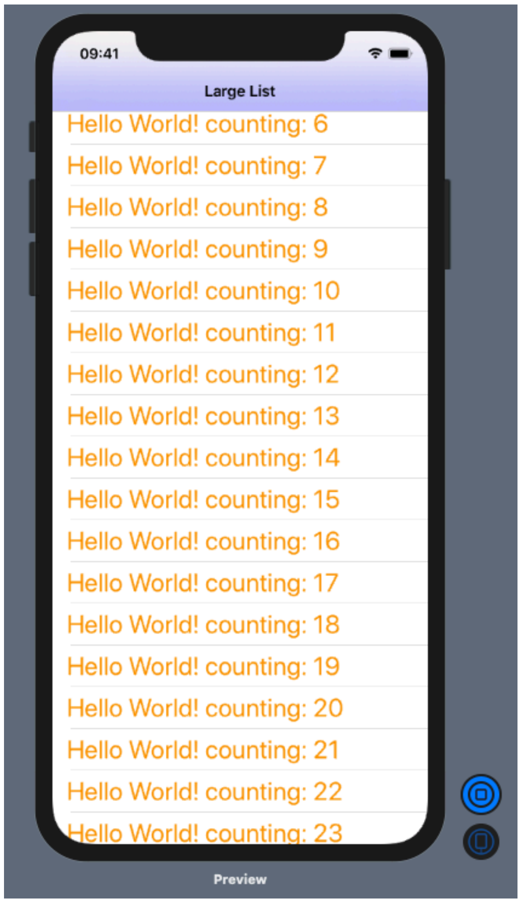
```
UINavigationBar.appearance().backgroundColor = .blue
```

In live preview:



Large List

Hello World! counting: 1
Hello World! counting: 2
Hello World! counting: 3
Hello World! counting: 4
Hello World! counting: 5
Hello World! counting: 6
Hello World! counting: 7
Hello World! counting: 8
Hello World! counting: 9
Hello World! counting: 10
Hello World! counting: 11
Hello World! counting: 12
Hello World! counting: 13
Hello World! counting: 14
Hello World! counting: 15
Hello World! counting: 16



Large List

Hello World! counting: 6
Hello World! counting: 7
Hello World! counting: 8
Hello World! counting: 9
Hello World! counting: 10
Hello World! counting: 11
Hello World! counting: 12
Hello World! counting: 13
Hello World! counting: 14
Hello World! counting: 15
Hello World! counting: 16
Hello World! counting: 17
Hello World! counting: 18
Hello World! counting: 19
Hello World! counting: 20
Hello World! counting: 21
Hello World! counting: 22
Hello World! counting: 23

See how the navigation bar changes as we scroll up.

3. List with Navigation and Editing

A `List` view can display data in a single column table. On its own a `List` has limited capability. By embedding it in a `Navigation` view we can enable a `List` view acquire capabilities such as adding new rows, deleting existing rows, and moving rows. To demonstrate, we create a project “List With Navigation And Editing”.

We are going to use a list of countries of the world as the data to populate a `List` view. So create the struct:

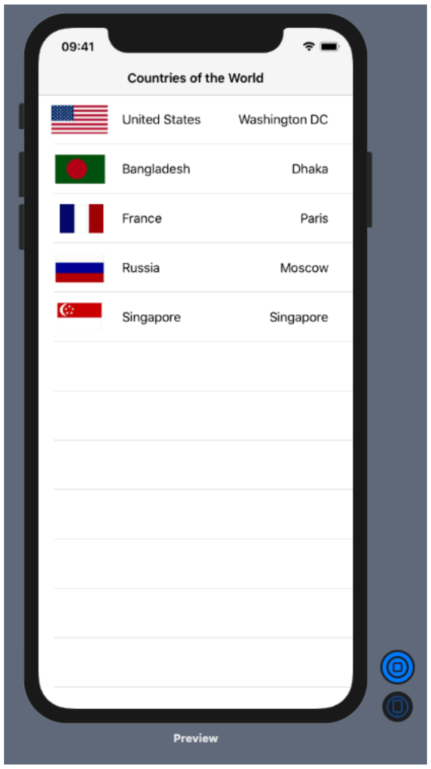
```
struct Country:Identifiable {
    var id = UUID()
    var flag:String
    var name: String
    var capital: String
    var description:String
    static func getCountries()-> [Country]{
        var countries = [Country]()

        countries.append(Country(flag:  "USA", name: "United States", capital: "Washington
               DC", description: "United States is a country in North America, a federal
               republic of 50 states. Besides the 48 conterminous states that occupy the
               middle latitudes of the continent, the United States includes the state of
               Alaska, at the northwestern extreme of North America, and the island state of
               Hawaii, in the mid-Pacific Ocean."))

        countries.append(Country(flag:  "Bangladesh", name: "Bangladesh", capital: "Dhaka",
               description: "Bangladesh is a country in Asia, located on eastern part of
                              Indian subcontinent beween West Bengal, Assam, and Tripura"))

        countries.append(Country(flag:  "France", name: "France", capital: "Paris",
               description: "France is a country in Europe. Historically and culturally among
               the most important nations in the Western world, France has also played a
               highly significant role in international affairs, with former colonies in every
               corner of the globe."))

        countries.append(Country(flag:  "Russia", name: "Russia", capital: "Moscow",
               description: "Russia is country that stretches over a vast expanse of eastern
               Europe and northern Asia. Once the preeminent republic of the Union of Soviet
               Socialist Republics (U.S.S.R.; commonly known as the Soviet Union), Russia
               became an independent country after the dissolution of the Soviet Union in
               December 1991."))

        countries.append(Country(flag:  "Singapore", name: "Singapore", capital: "Singapore",
               description: "Singapore, city-state located at the southern tip of the Malay
               Peninsula, about 85 miles (137 kilometres) north of the Equator. It consists of
               the diamond-shaped Singapore Island and some 60 small islets; the main island
               occupies all but about 18 square miles of this combined area. The main island
               is separated from Peninsular Malaysia to the north by Johor Strait, a narrow
               channel crossed by a road and rail causeway that is more than half a mile long.
                                                                                           "))

        return countries
    }
}
```

In the List view, we are going display the flag, name of the country, and capital. So we create the struct:

```
struct CountryRow: View {
    let flagName: String
    let name: String
    let capital:String
    var body: some View {
        HStack{
            Image(self.flagName)
               .resizable()
               .aspectRatio(contentMode: .fit)
               .frame(width: 76, height: 38, alignment: .center)
               .padding(.trailing, 10)
            Text(self.name)
               .font(.body)
            Spacer()
            Text(self.capital)
               .font(.body)
               .padding()
        }
    }
}
```

Then in the ContentView, we add the code:

```
struct ContentView: View {
    @State var countries = Country.getCountries()

    var body: some View {
        NavigationView{
            List(countries){country in
                CountryRow(flagName: country.flag, name:
                 country.name, capital: country.capital)
            }.navigationBarTitle("Countries of the
                   World", displayMode: .inline)
        }
    }
}
```



On the right we have shown a screen shot of live preview.

We have the `List` embedded in a `NavigationView` and we have a navigation bar. However, `List` view does not provide the view modifiers we need to implement editing. We have `ForEach` struct has the necessary view modifier methods for editing:

```
onDelete(perform:)
onMove(perform:)
```

So we modify our code to insert a `ForEach` loop:

```
struct ContentView: View {
    @State var countries = Country.getCountries()

    var body: some View {
        NavigationView{
            List{ForEach(countries){country in
                    CountryRow(flagName: country.flag, name: country.name,
                                                 capital: country.capital)
                }.navigationBarTitle("Countries of the World"
                                  , displayMode: .inline)
            }
        }
    }
}
```

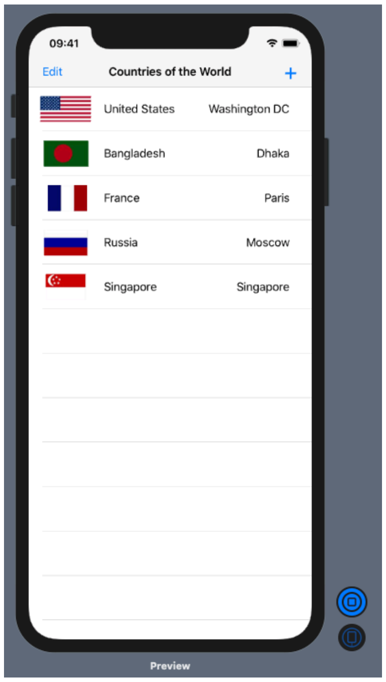
Next we are going to add onDelete and omMove modifiers. This would require adding the methods:

```
func delete(at offsets: IndexSet) {
    if let first = offsets.first {
         countries.remove(at: first)
    }
}
```

```
func move(from source: IndexSet, to destination: Int) {
     // sort the indexes low to high
     let reversedSource = source.sorted()
     // then loop from the back to avoid reordering problems
     for index in reversedSource.reversed() {
          // for each item, remove it and insert it at the destination
          countries.insert(countries.remove(at: index), at: destination)
     }
 }
```

We also need to a navigation bar item `EditButton()`. In addition, we are going to add another button for adding new records. Then the code in `body` becomes:

```
var body: some View {
        NavigationView{
            List{
               ForEach(countries){country in
                   CountryRow(flagName: country.flag,
                              name: country.name,
                              capital: country.capital)
               }.onDelete(perform: delete)
                .onMove(perform: move)
            }.navigationBarTitle("Countries of the World"
                                              , displayMode: .inline)
            .navigationBarItems(leading: EditButton(),
                                trailing:Button( action:
                                                   add) {
                                    Text("+")
                                    .font(.largeTitle)
             })
        }
    }
```



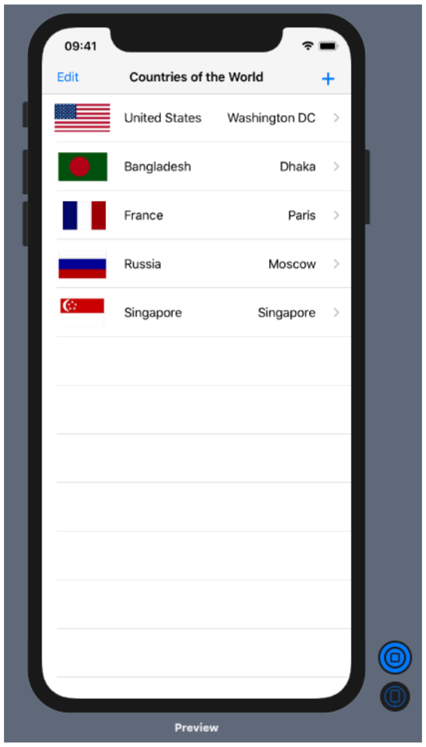
To add a new item to the list, all we have to do is add a new element to the countries array and SwiftUI will automatically take care of updating the `List` view. Here, we just add two items manually:

```
func add(){
     countries.append(Country(flag:  "UK", name: "United Kindgom", capital: "London",
          description: "United Kinddom island country located off the northwestern coast of
                         mainland Europe. The United Kingdom comprises the whole of the island
                         of Great Britain—which contains England, Wales, and Scotland—as well as
                         the northern portion of the island of Ireland. The name Britain is
                         sometimes used to refer to the United Kingdom as a whole. "))

    countries.append(Country(flag:  "Canada", name: "Canada", capital: "Ottawa",
          description: "Canada is second largest country in the world in area (after Russia),
                         occupying roughly the northern two-fifths of the continent of North
                         America. Despite Canada's great size, it is one of the world's most
                                                                sparsely populated countries. "))

      }
```

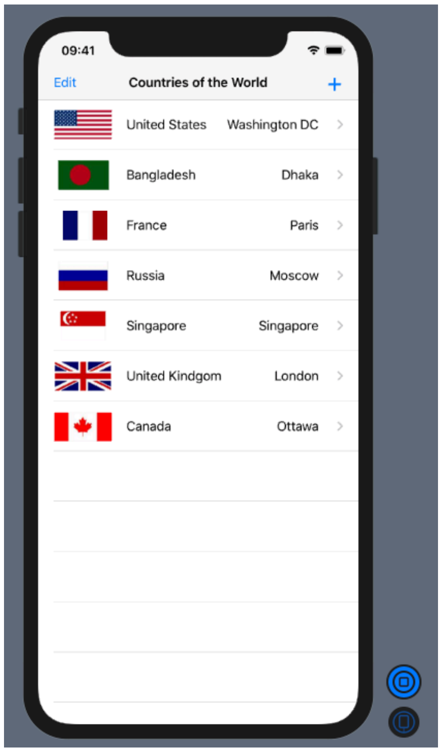
We also want to navigate to a detail view when we tap on a row. This can be achieved by embedding the `ListRow` in a `NavigationLink`. The final code of `body` in `ContentView` is:

```
var body: some View {
    NavigationView{
       List{
          ForEach(countries){country in
              NavigationLink(destination: CountryDetail(country: country)){
                     CountryRow(flagName: country.flag, name: country.name,
                                                   capital: country.capital)}
              }.onDelete(perform: delete)
               .onMove(perform: move)
       }.navigationBarTitle("Countries of the World",displayMode:.inline)
        .navigationBarItems(leading: EditButton(),trailing:Button(action: add){
                                   Text("+")
                                      .font(.largeTitle)})
     }
 }
```

In live preview, when we run the app and tap the + button once:

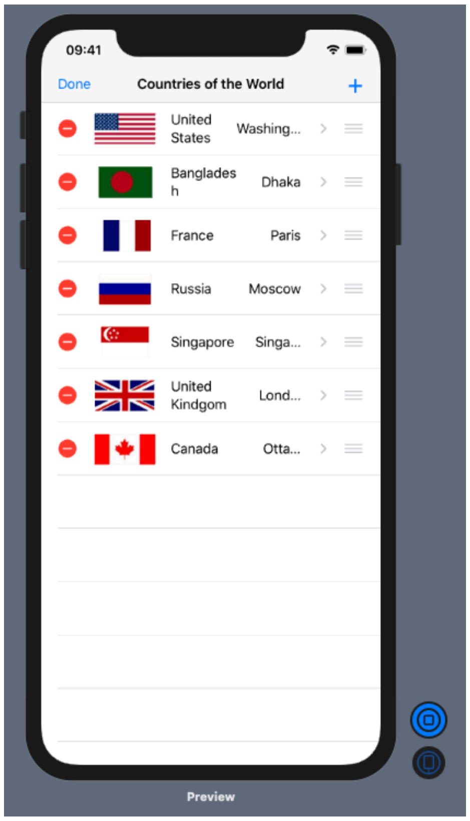


United States	Washington DC
Bangladesh	Dhaka
France	Paris
Russia	Moscow
Singapore	Singapore



United States	Washington DC
Bangladesh	Dhaka
France	Paris
Russia	Moscow
Singapore	Singapore
United Kindgom	London
Canada	Ottawa

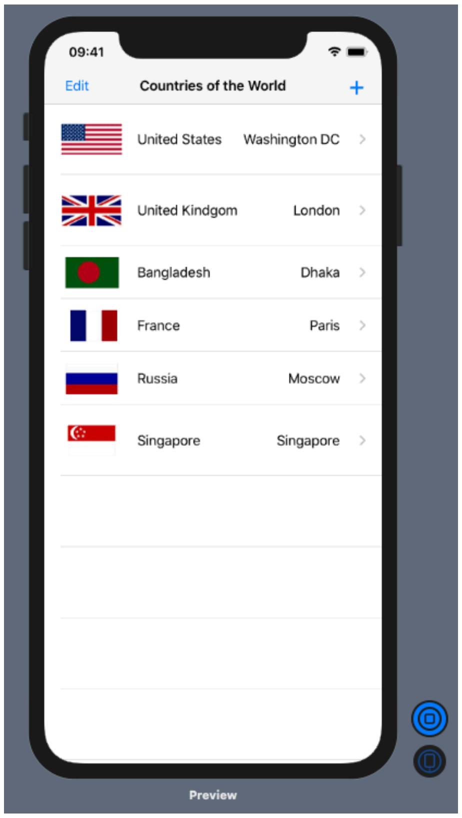
Tapping the Edit button will bring up the edit option. By choosing the delete option on the left of row and then tapping the delete button on the right, we can delete that row. The move option on the right can be dragged to move a row.


09:41

Done Countries of the World +

United States	Washing...
Bangladesh	Dhaka
France	Paris
Russia	Moscow
Singapore	Singa...
United Kindgom	Lond...
Canada	Otta...

Preview


09:41

Edit Countries of the World +

United States	Washington DC
United Kindgom	London
Bangladesh	Dhaka
France	Paris
Russia	Moscow
Singapore	Singapore

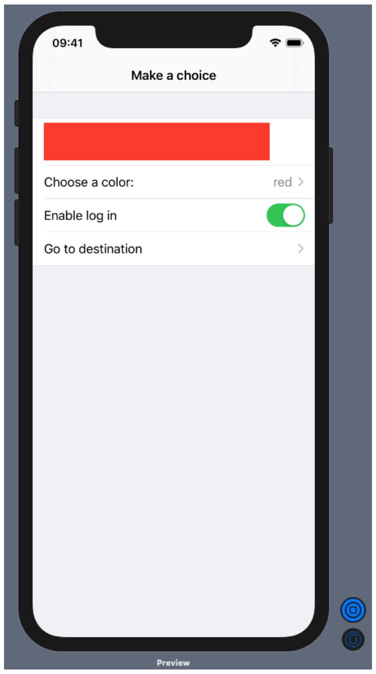
Preview

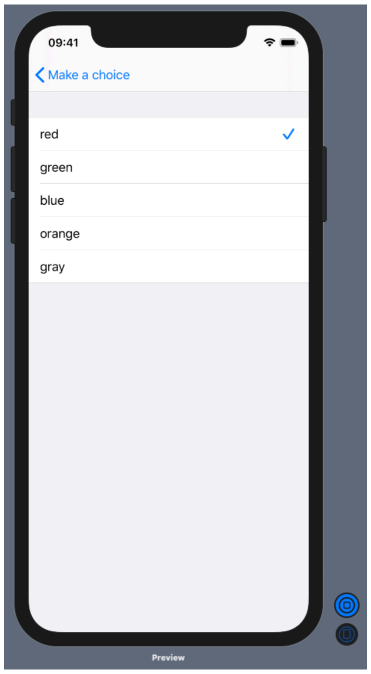
4. Form with Navigation

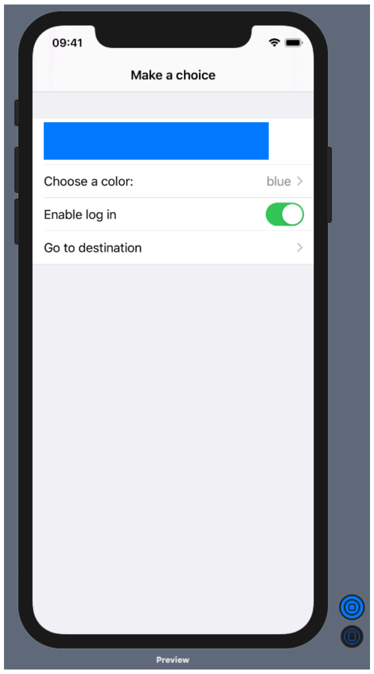
To enable navigation from within a `Form`, we have to embed the `Form` in a `NavigationView`. To demonstrate, we create a project “Form with Navigation Demo”. The code we put in is quite similar to a project in the previous chapter on `Form`:

```
struct ContentView: View {
    let colors: [Color] = [.red, .green, .blue, .orange, .gray]
    @State var selectedColor = 0
    @State var choice = true

    var body: some View {

        NavigationView{
            Form{
                Rectangle()
                    .fill(colors[selectedColor])
                    .frame(width:300,height: 50)

                Picker(selection: $selectedColor, label: Text("Choose a color:"))
                {
                    ForEach(0..<colors.count) {i in
                        Text(self.colors[i].description).tag(i)

                    }
                }.pickerStyle(DefaultPickerStyle())
                Toggle(isOn: $choice){
                    Text("Enable log in")
                }
                if choice {
                    NavigationLink(destination: DetailView1()
                        .navigationBarTitle("DetailView1", displayMode: .automatic))
                    {
                        Text("Go to destination")
                    }
                }
                else{
                    NavigationLink(destination: DetailView2()
                        .navigationBarTitle("DetailView2", displayMode: .automatic))
                    {
                        Text("Go to destination")
                    }
                }

            }.navigationBarTitle("Make a choice", displayMode: .inline)
        }
    }
}
```

Inside the `Form`, we have a `Rectangle` whose background color is chosen by a `Picker` control. The `Picker` control with a `DefaultPickerStyle` has a different appearance.

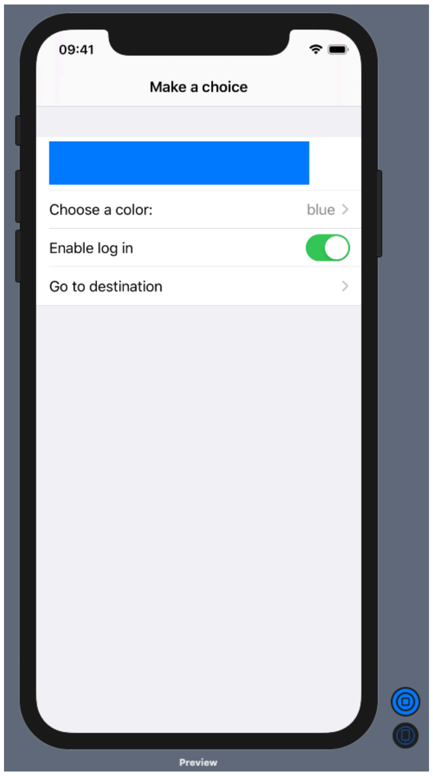






Tapping the `Picker` opens a view with the options available in the `Picker`. Choosing a different option changes the background color of the rectangle.

Next depending on the Toggle position, the navigation link destination will be different:

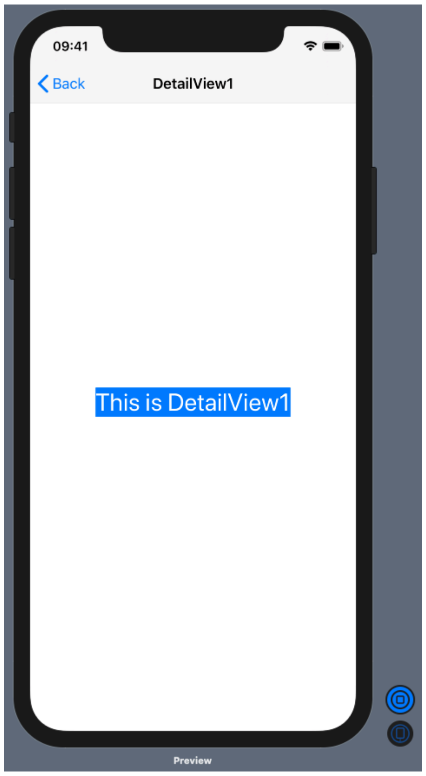

09:41

Make a choice

Choose a color: blue

Enable log in

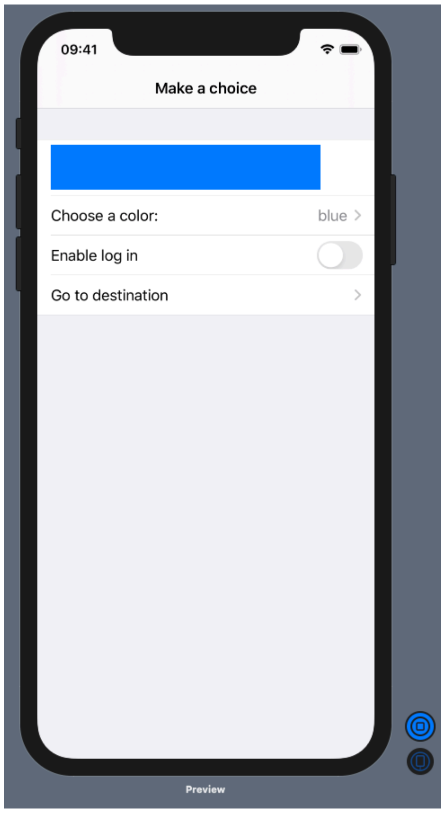
Go to destination


09:41

Back

DetailView1

This is DetailView1

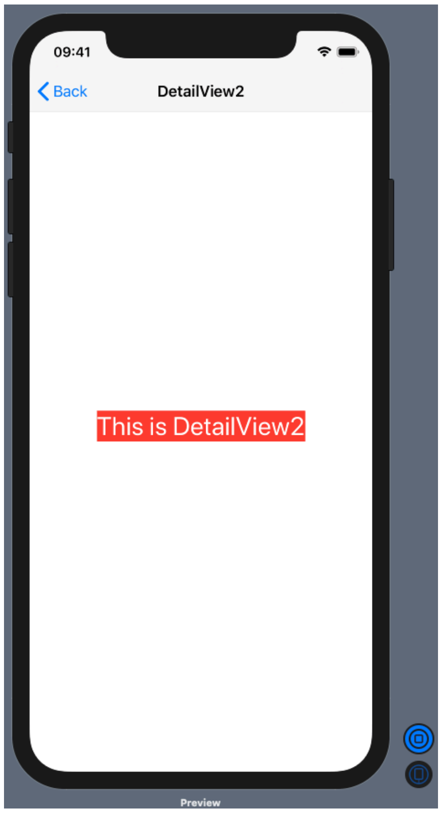

09:41

Make a choice

Choose a color: blue

Enable log in

Go to destination


09:41

Back

DetailView2

This is DetailView2

5. TabView

SwiftUI provides us with `TabbedView` to create apps with tabs just like UITabBar in UIKit. This allows us to switch between different views in our app. The `TabView` has two initializers:

```
init(@ViewBuilder content: () -> Content)
```

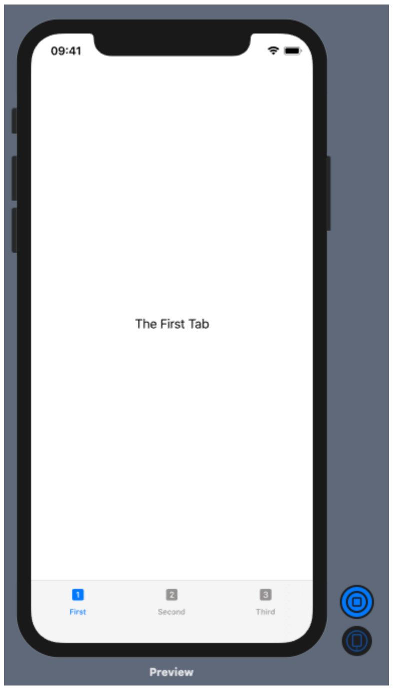
This is available when `SelectionValue` is `Int` and the first tab is selected by default

```
init(selection: Binding<SelectionValue>?, @ViewBuilder content: () -> Content)
```

Creates an instance that selects from content associated with Selection values.

To demonstrate the use of the first initializer, we create a project “Simple TabbedView Demo” This is very similar to the example in Apple documentation:

```
struct ContentView: View {
    var body: some View {
         TabView {
            Text("The First Tab")
                .tabItem {
                     Image(systemName: "1.square.fill")
                     Text("First")
                }
           Text("The Second Tab")
                .tabItem {
                     Image(systemName: "2.square.fill")
                     Text("Second")
                }
          Text("The Third Tab")
              .tabItem {
                     Image(systemName: "3.square.fill")
                     Text("Third")
            }
        }
    }
}
```



To add a `View` to a `TabView`, we embed it in the `TabView`. For each View in a `TabView`, we apply the `tabItem(_:)` view modifier:

```
func tabItem<V>(@ViewBuilder _ label: () -> V) -> some View where V : View
```

We provide each tab item with an image and title. If we want to control which tab is programmatically active, we add a tag to the item. For that we need to use the second initializer.

To demonstrate, we create a project “TabView with Tag Demo”. In our previous project, we didn’t use tag explicitly. This implies that SwiftUI used tag in the background. Presumably, using integer starting from 0. We can still use integers by adding `.tag(0)` etc to our tab items. Of course, remembering the tags must be unique, repeating the same value for more than one tab is not permitted. Instead of using integers directly, we use more user friendly tags by introducing an enum to ContentView:

```
extension ContentView{
    enum TAB{
        case red
        case blue
        case green
    }
}
```

We need to add a State property for the selected tab to ContentView:

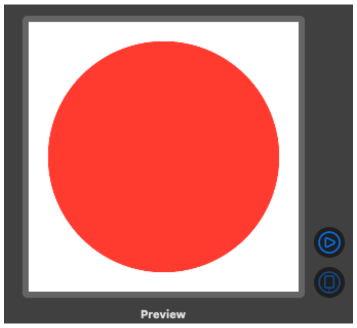
```
@State private var selectedTab = TAB.red
```

We add three placeholder files `RedView`, `BlueView`, and `GreenView` to be used in the `TabView`. The code in ContentView is

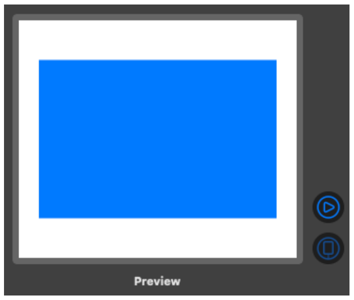
```
struct ContentView: View {
    @State private var selectedTab = TAB.red

    var body: some View {
        TabView(selection: $selectedTab) {
            RedView()
                .onTapGesture {
                    self.selectedTab = TAB.green
                }
                .tabItem {
                    Image(systemName: "1.circle")
                    Text("RedView")
                }
                .tag(TAB.red)

            BlueView()
                .tabItem {
                    Image(systemName: "2.circle")
                    Text("BlueView")
                }
                .tag(TAB.blue)

            GreenView()
                .tabItem {
                    Image(systemName: "2.circle")
                    Text("GreenView")
                }
                .tag(TAB.green)
        }
    }
}
```

We can check that the code is working as expected. When we tap on the `RedView()`, the selected index is changed and the tab with `GreenView()` is shown. Now it is time to replace the view placeholders with some code.

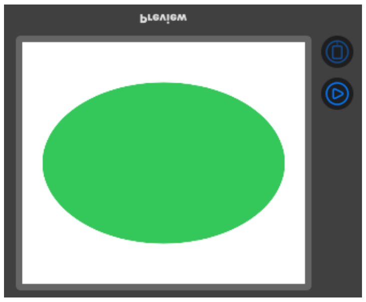
```
struct RedView: View {
    var body: some View {
        Circle()
            .fill(Color.red)
        .frame(width: 300, height: 300)
    }
}
```



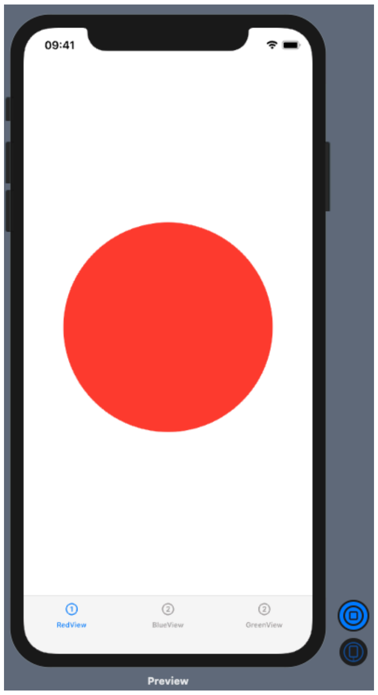
```
struct BlueView: View {
    var body: some View {
        Rectangle()
            .fill(Color.blue)
        .frame(width: 300, height: 200)
    }
}
```

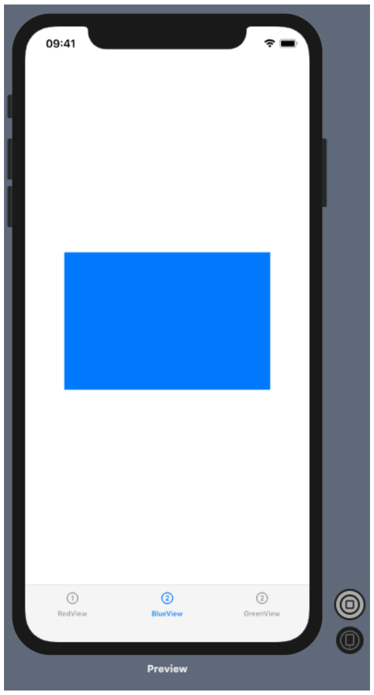


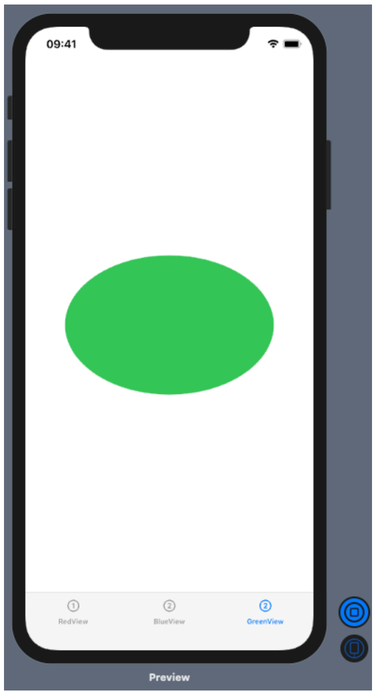
```
struct GreenView: View {
    var body: some View {
        Ellipse()
            .fill(Color.green)
        .frame(width: 300, height: 200)
    }
}
```



The in live preview:







When we tap on the red circle in the first tab, the third tab with the green ellipse is shown.

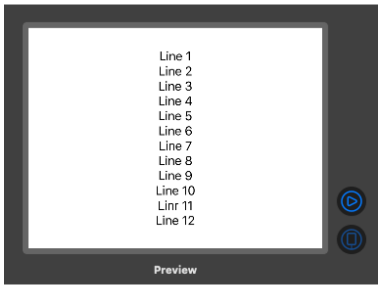
6. Grouping Views

There is a limit to how many views you can embed in a `VStack`, `HStack`, or `ZStack` – it is 10. For example, this would work fine:

```
VStack {
     Text("Line 1")
     Text("Line 2")
     Text("Line 3")
     Text("Line 4")
     Text("Line 5")
     Text("Line 6")
     Text("Line 7")
     Text("Line 8")
     Text("Line 9")
     Text("Line 10")
}
```

If we attempt to add another view, there will be error. SwiftUI provides a solution to this problem – we create groups and put 10 or fewer views in each group. The following works fine:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Group{
                Text("Line 1")
                Text("Line 2")
                Text("Line 3")
                Text("Line 4")
                Text("Line 5")
                Text("Line 6")
                Text("Line 7")
            }
            Group{
                Text("Line 8")
                Text("Line 9")
                Text("Line 10")
                Text("Linr 11")
                Text("Line 12")
            }
        }
    }
}
```



```
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Linr 11
Line 12
```

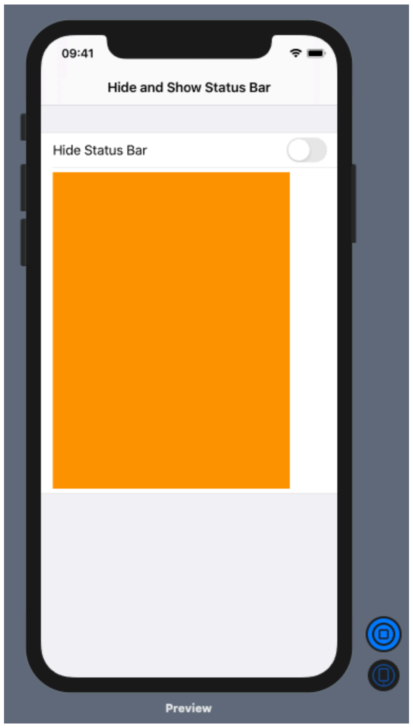
Note that members of a group acts like one – for example, they transition together.

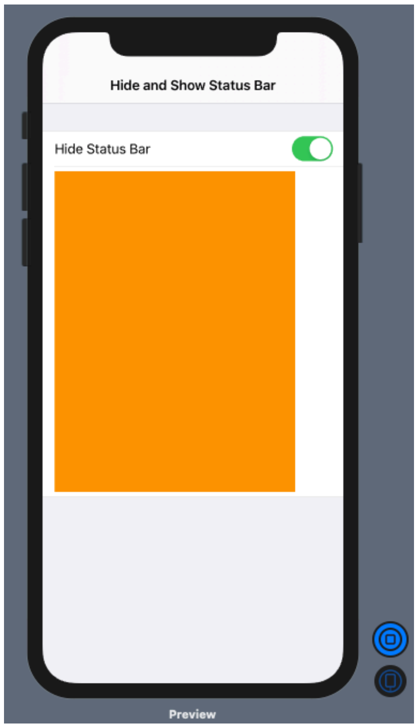
7. Hide and Show Status Bar

We can hide or show the status bar of a view by applying the view modifier `.statusBar(hidden:)` to the view. To demonstrate, we create a project "Hide and Show Status Bar Demo" with the code:

```
struct ContentView: View {
    @State var flag = false
    var body: some View {
        NavigationView{
            Form{
                Toggle(isOn: $flag){
                    Text("Hide Status Bar")
                }
                Rectangle()
                    .fill(Color.orange)
                    .frame(width: 300, height: 400)
            }.navigationBarTitle("Hide and Show Status Bar", displayMode: .inline)
        }.statusBar(hidden: flag)
    }
}
```

Then in live preview:





8 ScrollView

When the content is too large to fit on the screen of a device, we need to be able to scroll to view any portion off the screen. SwiftUI provides us with `ScrollView`, any view want to be scrollable should be embedded in a `ScrollView`. The initializer proved for `ScrollView` is:

```
init(_ axes: Axis.Set = .vertical, showsIndicators: Bool = true, @ViewBuilder content: ()
                                                                        -> Content)
```

The parameters of the initializer are

`axes`

The scroll view's scrollable axis. The default axis is the vertical axis.

`showsIndicators`

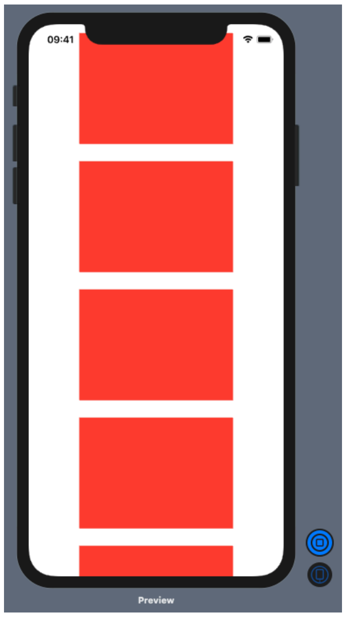
A Boolean value that indicates whether the scroll view displays the scrollable component of the content offset, in a way suitable for the platform. The default value for this parameter is true.

`content`

The view builder that creates the scrollable view.

To demonstrate, we create the project “ScrollView Demo” based on the TabView App template. We need to add a third tab to the default `TabView` with two tabs. The first tab will demonstrate vertical scroll. So, we add the code:

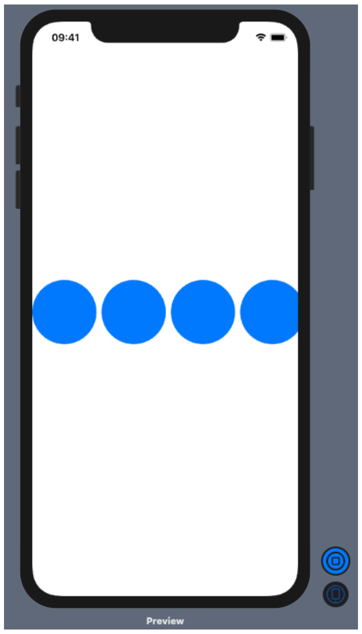
```
struct VerticalScroll: View {
    var body: some View {
        ScrollView{
            ForEach(1...10, id:\.self){_ in
                Rectangle()
                    .fill(Color.red)
                    .frame(width: 250, height: 180)
                    .padding(10)
            }
        }
    }
}
```



We can check in live preview (in a simulator or device) we can scroll vertically. We use `VerticalScroll` for the first tab in the `TabView`. Note that vertical scroll is default. So, it can be omitted in the initializer

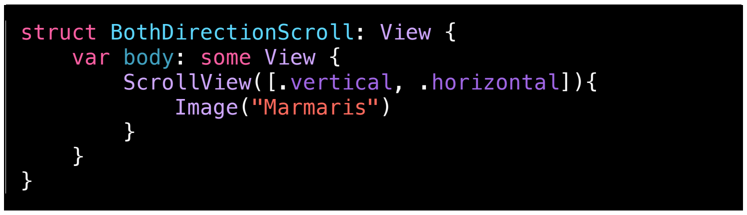
The second tab demonstrates horizontal scroll. We add the following code:

```
struct HorizontalScroll: View {
    var body: some View {
        ScrollView(.horizontal){
            HStack{
                ForEach(1...10, id:\.self){_ in
                    Circle()
                       .fill(Color.blue)
                      .frame(width: 100, height: 100)
                }
            }
        }
    }
}
```

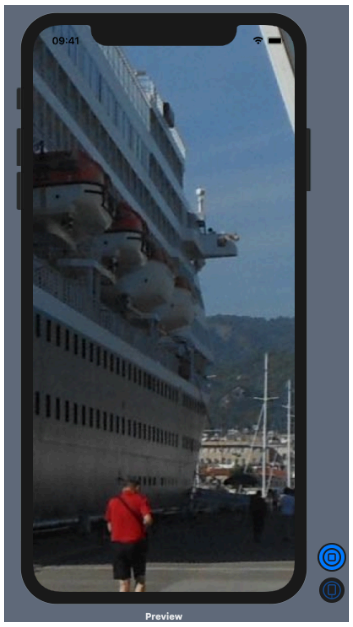


Now we can scroll horizontally.

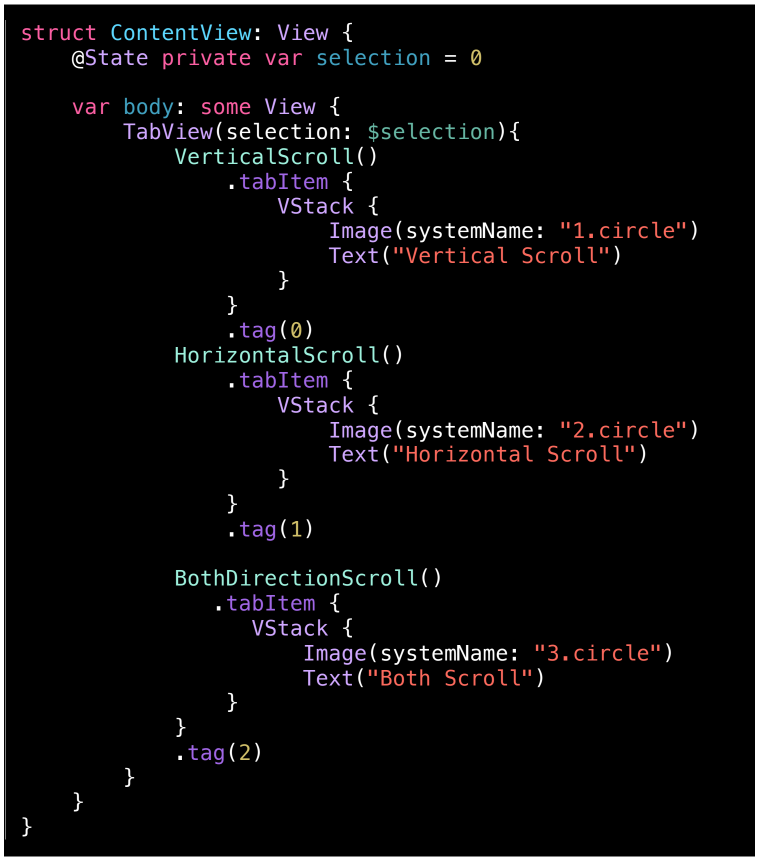
In the third tab, we load a large image and scroll capability in both vertical and horizontal directions.



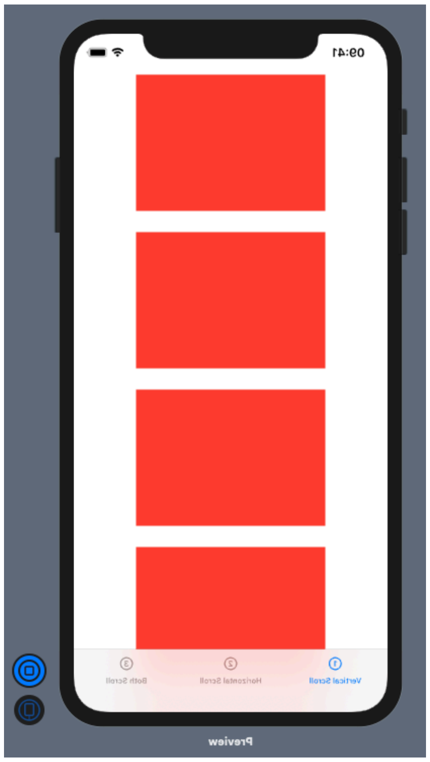
```swift
struct BothDirectionScroll: View {
    var body: some View {
        ScrollView([.vertical, .horizontal]){
            Image("Marmaris")
        }
    }
}
```



The code in ContentView is:



```swift
struct ContentView: View {
    @State private var selection = 0

    var body: some View {
        TabView(selection: $selection){
            VerticalScroll()
                .tabItem {
                    VStack {
                        Image(systemName: "1.circle")
                        Text("Vertical Scroll")
                    }
                }
                .tag(0)
            HorizontalScroll()
                .tabItem {
                    VStack {
                        Image(systemName: "2.circle")
                        Text("Horizontal Scroll")
                    }
                }
                .tag(1)

            BothDirectionScroll()
               .tabItem {
                   VStack {
                       Image(systemName: "3.circle")
                       Text("Both Scroll")
                   }
               }
               .tag(2)
        }
    }
}
```



We run in live preview or on a device (simulator) and check everything as designed.